Version NISTUGV0.2
January 14, 2002

NIST DASE DEVELOPMENT ENVIRONMENT

USER'S GUIDE

1 INTRODUCTION

This document describes the use of the NIST DASE Development Environment (NIST DASE DE). It is composed of a Set-top Box (STB) simulation, a PAE prototype implementation, example applications, and tools. The purpose of the Simulation is to allow the execution of Digital TV Application Software Environment (DASE) programs. These programs use the DASE Application Programming Interface (API) to retrieve data from the Simulation. The DASE API communicates with the simulation via a Hardware Abstraction Layer (HAL) in order to isolate the API from the underlying system functionality and data management.

The User's Guide describes the components of the simulation and instructions on how to install, compile, and run the simulation. In addition it describes the platform requirements, utility programs, the java runtime environment extensions, the NIST bit stream syntax and example application use cases, and helpful tips and examples for using the simulation.

2 VERSION AND USE INFORMATION

This version of the User's Guide corresponds to the NIST DASE Development Environment distribution labeled NISTDEV0.2. The name of the uncompressed download file is nistdeV0.2.tar. The instructions for installation and usage apply to the Solaris and Linux platforms. Hints for porting the environment to a Windows platform are given in sections C.3.

3 TERMS AND CONVENTIONS

ATSC Data	A file containing simulated ATSC transport stream data. The file is used as input for the NIST STB simulation. The file is created by the Data Generator program and is designated with `.atsc` file extension.
DASE Application	An application that uses some of the PAE APIs, but not limited to them. A receiver resident application (A native EPG is an example).
Data Generator	A utility program used to create simulated ATSC DATA.
DE	Development Environment, the NIST DASE Development Environment that includes among other things the prototype PAE API implementation, ATSC STB simulation, and sample X-lets.
DTV Simulator	An application that controls the functions in the STB simulation. A simulated GUI Remote Control is used for input and a simulated GUI DTV Display is used for output.
Feeder	A GUI utility program that can be used to input pre-determined ATSC DATA files into the STB simulation.
HAL	Hardware Abstraction Layer; The HAL is a layer of software that isolates the API from the underlying STB functionality.
JavaSTBMain	The java class file used to start (run) the NIST STB Simulation.
JREx	Java Runtime Environment Extensions
PCR Manager	Provides a program clock reference to the HAL JMF player
STB Simulation	The NIST STB Simulation encapsulates the functionality of an ATSC STB necessary to implement the DASE PAE.
X-let	An application that adheres to the X-let specification.

In the examples that follow, certain shell variables are used to facilitate explanations.
The table below list example shell variables with pre-defined values.

Shell Variable	Description	Example
HOME	Home directory of user	/export/home/rob
ROOTDIR	Root directory of the NIST DASE DE	/export/home/rob/nist_ri
RUNTIME	Directory containing compiled java and c files	$ROOTDIR/simulation/runtime
JAVA_HOME	Location of Java Distribution	/usr/java

4 COMPONENTS OF THE SIMULATION

The Simulation is comprised of several components, the majority is written in the Java language and some is written in the ‘C’ language. The Java components, existing as Java byte code libraries, are portable to any platform supporting a Java virtual machine. The ‘C’ components are compiled for the host system

4.1 Native Libraries

The “native” libraries consist of support functions written in the ‘C’ language.

The library `libjreX.so` contains bridging functions that allow the replacement `java.io` classes to communicate to the underlying native Java IO implementation. The location of this library must be found on the path set in the `LD_LIBRARY_PATH` when the Java runtime environment (JRE) extensions are used. These extensions are part of the NIST prototype implementation and are modifications to the Java runtime environment (JRE) necessary to use the Data Carousel API. ANNEX A describes the JRE extensions in detail. After compilation the `libjreX.so` library is placed in the `$HOME/nist_ri/simulation/runtime` directory.

Another ‘C’ program is the PCR Manager, which provides a program clock reference to the HAL JMF player. This program, `PCRManager`, is part of the NIST DE distribution and must be in the user’s path. The build system places `PCRManager` in the `nist_ri/bin` directory. Update: the PCRManager source code is in the distribution, but has been removed from the build process. The PCRManager can be used to synchronize simulated video. No action is necessary if you don’t want this feature.

4.2 Java Libraries

The core of the STB simulation is implemented in Java. All of the DASE API and the HAL are implemented in Java and delivered as Jar files. After executing the build process for the NIST prototype implementation you will have these Jar files (under the `$HOME/nist_ri/simulation/runtime` directory):

- **dase.jar** The DASE API and NIST hardware abstraction layer (HAL)
- **stb.jar** The STB simulation classes
- **jreX.jar** Contains classes that replace some `java.io` classes in order to provide Carousel file reading. Only built when JRE extensions are enabled in the configuration step.
- **devkit.jar** Contains utilities, tools, example DASE applications and X-lets.

Other components needed are the Java runtime environment jar files and the Java Media Framework (JMF) jar files. The NIST DE has been developed and built using the Java Development Kit (JDK) and JMF toolkits from Sun Microsystems.

4.3 ATSC Simulation Data

Sample simulation atsc data files are located under the `$HOME/nist_ri/simulation/data` directory. These files are used to statically initialize the STB simulation with atsc transport stream data. The atsc files are static representations of atsc table information. These files are used when running the simulation with the command line interface. See the section 5.3.2 for details. You can also create your own test data (atsc files) with the Data Generator utility. See section 8.

5 INSTALLATION AND USAGE

5.1 Environment Setup

5.1.1 Setting the CLASSPATH Environment Variable

The environment variable `CLASSPATH` must be set prior to running the Simulation. This variable is used by the Simulation to find the Jar files for the NIST DE and is passed to the Java virtual machine created by the Simulation controlling process. The Jar files for the Java Media Framework must also be located on this path.

An example `CLASSPATH` would be set using *csh* as follows:

```
setenv CLASSPATH ${HOME}/nist_ri/simulation/runtime/stb.jar:\

${HOME}/nist_ri/simulation/runtime/dase.jar:\

${HOME}/nist_ri/simulation/runtime/devkit.jar:/usr/java/jre/lib/jmf.jar
```

If the Simulation does not find the Java runtime libraries they must be added to the `CLASSPATH` as well. For the Sun JDK 1.2, they are usually located in `usr/java/jre/lib/rt.jar`. However, the Sun JVM will usually find these libraries.

Examples of .cshrc and .bashrc files can be found in Annex E.

5.1.2 Setting options for the Java command

In order to use the JREx functionality, certain options need to be set for the Java command. It is recommended that you set up the following alias and use the alias to start the simulation (see section 5.3.1). Here is a csh example.

```
setenv RUNTIME ${ROOTDIR}/simulation/runtime
alias j 'java -Xbootclasspath:${RUNTIME}/jreX.jar: \
${JAVA_HOME}/jre/lib/rt.jar:${CLASSPATH} -
Dsun.boot.library.path=${JAVA_HOME}/jre/lib/i386:${RUNTIME}'
```

Examples of .cshrc and .bashrc files can be found in Annex E.

5.2 Installing and Compiling

The build/make process has been tested under both Solaris and Linux.

1. Un-archive the distribution "tar" file (nistdeV0.1.tar) by typing:

   % tar -xvf nistde.V0.1.tar

   This creates a subdirectory called 'nist_ri' in the current directory. The full path of the 'nist_ri' directory is referred to as 'ROOTDIR' in this document.

2. Change your current directory to 'nist_ri' by typing:

   % cd $ROOTDIR

3. Configure the NIST RI for your system by typing:

   % ./configure

   It is assumed here that you will build the NIST RI on the same system that you are using to run configure.

Note, configure must be able to correctly locate where the JDK and JMF are installed. These locations will vary based on the local system, and can be specified to configure directly (if necessary) through the use of command line options to configure:

--with-jdk=<path> Specify the location of the JDK

--with-jmf=<path> Specify the location of the JMF

Also, if Java runtime environment extensions to enable Carousel file reading are available (not all distributions of the NIST DASE DE include this), they can be optionally configured for compilation using:

--with-jrex Include JRE extensions in the compilation

For most installations, the configure command will be:

% ./configure –with-jrex

In a public distribution of the NIST DASE DE, you may have to rewrite some of the java.io classes. For each class, an exhaustive list of changes is in the tree ($ROOTDIR/ext/jreX/javalib/changes/*.txt), based on the JDK 1.2.2 sources. In order to run Xlets that access the Data Carousel (e.g., the Stock Ticker), you need the jre extensions.

Following these lists, you will write each class and put the new source code under $ROOTDIR/ext/jreX/javalib/src.

4. Compile the NIST DASE DE on your system by typing:

% make

If successful, the api, simulation and application jar files (**dase.jar, stb.jar & devkit.jar**) will be installed to the '$ROOTDIR/simulation/runtime' subdirectory. (If the --with-jrex option was specified to configure, the jar file **jreX.jar** and the library file **libjreX.so** will be installed there as well) In addition, the JMF test program, PCRManager, will be installed in the '$ROOTDIR/bin' subdirectory (Update: not unless to explicitly include the PCRManager into the build process. Currently it is removed.)

To restore (clean) the distribution tree to its original state use “make distclean”. To rebuild, follow steps 3 and 4 above.

% cd $ROOTDIR

% make distclean

Note: On some systems, a potential compiling problem exists, where “java tools.simulation.DataGen –c …” hangs. The solution is unknown, however, some work-a-rounds appear to work: 1) kill the compile, and retype make, 2) kill the compile, do a make distclean, and then a make. In any event, at this point the implementation has compiled, the “DataGen” program is creating data files for the demonstration and is not necessary needed to run the simulation.

5.3 Running the Simulation

Now that the appropriate environment variables have been set and the NIST DE has been compiled, you are ready to run the Simulation. The execution of the Simulation will be described first, followed by a specific example on starting the Simulation and executing a DASE application.

The NIST DE should have been installed in a location where the user has write permission to the file system. In the examples that follow, it is assumed that the NIST DE has been installed in the user's home directory ($HOME).

5.3.1 JavaSTBMain Usage

The *root* application in the simulation is JavaSTBMain. To start the Simulation, run the Java program **gov.nist.hwabstract.JavaSTBMain**. The syntax is:

Usage: java gov.nist.hwabstract.JavaSTBMain [-l] [-r <fifoName> | -r <fileName>] [-s] [-pcr] [-t <n>] [-h | -help]

Note in order to run Xlets using the Java Runtime Environment extensions (jreX), replace 'java' with the 'j' alias defined in ANNEX E. It is highly recommended that you use the j alias in all cases.

Option	Synopsis
-l	Logs to several .logs files
-r	Read ATSC data from a regular file
-f	Read ATSC data from a FIFO file
-s	Auto start the DtvSimulator and the DTV Display
-pcr	Starts a PCRManager
-t	Create and use <n> FIFOs for controlling <n> tuners (same path as <fifoName>/<fileName>
-h or -help	Prints help information
Default	-f $HOME/inputFIFO

`JavaSTBMain` will create a data FIFO named `inputFIFO` in the current directory if the `-f` option is not used. If the `–f` option is used; the name of the FIFO must also be given. Similarly, when the `-r` option is used, the name of a regular file must be given on the command line. These options are mutually exclusive and the last option given will take effect.

`JavaSTBMain` initializes the Java virtual machine and creates the STB Simulation manager thread inside the JVM. If all goes well, a prompt will appear allowing you to type in the name of a Java application to run. All of the class files for this application must be found on the `CLASSPATH`. There are several example applications in the `devkit` directory of the distribution. The applications are placed in the Jar file `devkit.jar` located in the `nist_ri/simulation/runtime` directory.

The –t option allows you to select the number of tuners for the STB simulation. Support for multiple tuners at this time is limited. It is recommended that only one tuner be used at present.

The `-s` option causes the Simulation controller to start up the `applications/native/dtvsimulator/DtvSimulator` object located in `devkit.jar`. This object is the graphical controller for the DTV simulator, allowing startup of the EPG and other native applications from a simulated remote control. This option also turns on the DTV display window. A detailed description of the DTV Simulator application can be found in section 7.

The name of the application to run must be fully qualified. For example, to run the application `DtvSimulator` from package `applications.nativeapps.dtvsimulator`, then the name you need to enter on the Simulation command line is `applications.nativeapps.dtvsimulator.DtvSimulator`.

If the application class cannot be found, you will receive the message `Can't find class for application` *`'appname'`*. You will also receive this message if any of the classes that are used by the application cannot be found.

After the application is started, you are prompted for another application to run. By entering '+', you can run the previous application again (e.g. a new instance of the same class). However, note that the reappearance of the prompt does not mean that the application has finished. If the application creates threads, those may still be running.

Optional logging may be turned on by using the `–l` option with Java`STBMain`. The messages from Java`STBMain` and the JVM are stored in the log file `Java_STBMain.log`. Messages from the Java simulation objects are stored in `Java_STBSim.log`, while HAL messages are stored in file `Java_HWAbstract.log`. These log files are created in the directory where Java`STBMain` is run. Additionally, all error messages are written into the corresponding log file if logging is selected. With no logging those selected error messages go to `stderr`.

To end the simulation, type `q` on the command line. Alternatively, pressing **CTRL-C** will terminate the simulation cleanly. If the simulation is terminated with a SIGTERM signal from another process it will shutdown cleanly. However, if a SIGKILL signal is sent to the `JavaSTBMain` process (using the '`kill –9`' command), then the `PCRManager` process (if started) will not be terminated. This process must then be terminated independently of the Java`STBMain` process.

There are two ways to run Xlets in the simulation. The first way is via a command line interface built into the simulation, called RunXlet. The second method is via a native DTV Simulator with a Remote Control application. Details of these options are found in section 6 and section 7.

5.3.2 Feeding Data Into the Simulation

The STB Simulation expects to receive a data stream containing ATSC tables. This stream is fed to the Simulation over a FIFO created by the Simulation. By default, the FIFO is created in the users $HOME directory, and is called `inputFIFO`. By using the `-f` parameter to `JavaSTBMain`, you can change the location of the FIFO. However, you must have the appropriate file permissions to create the FIFO.

Here's an example of how to run the Simulation with complete logging and specifying the location of the FIFO:

```
% j gov.nist.hwabstract.JavaSTBMain -l -f $HOME/inputFIFO
```

Any feeder program can be used as long as the data stream is compatible with the Simulation. The output of the data feeder should be redirected to the FIFO used by the Simulation, as discussed above. Appendix Bitstream Syntax describes the bit stream syntax.

Example data streams are located in the `nist_ri/simulation/data` directory of the distribution. Data can be fed into the Simulation by sending it directly to the FIFO from the data file by using the `cat` command in Unix. For example, if the data file is `simple.atsc` then use this command:

```
% cat $ROOTDIR/simulation/data/simple.atsc > $HOME/inputFIFO
```

The alternative method to inject data into the Simulation is to have the data read from a single file by specifying the `-r` parameter to `JavaSTBMain`. The same data file sent into a FIFO can be read directly by using this command:

```
% j gov.nist.hwabstract.JavaSTBMain -r $ROOTDIR/simulation/data/simple.atsc
```

Using this method saves time and removes the extra step of sending the data into a FIFO. However, only the data read at startup can be used; no updates are possible.

The utility program called Feeder provides another way to send data into the simulation. This program has a graphical interface and can be set with pre-defined data files. The Feeder program is used in the Dtv Simulator demonstration. See sections 7 and 9 for details.

There is a program in the `devkit.jar` library called `ShowATSCTables` that will dump the contents of the current data tables. To run this program, type `tools.simulation.ShowATSCTables` on the Simulation command line.

5.3.3 Simulation State Data

When the Simulation is started, it attempts to locate the file `STBSim.dat` in the current directory. This file contains the data used by the Simulation to maintain information about the STB environment. Users, common references and other environment information are contained in this file. If this file is not present, the Simulation will print a warning message, but execution will continue.

There is a program in the `devkit.jar` library called `CreateSTBDatabase`. This program is executed by entering `tools/simulation/CreateSTBDatabase` on the simulation command line. This program creates several users along with their preferences, the common preferences, and some STB settings. The Simulation manager saves these settings into the file `STBSim.dat`. The next time the Simulation is started these settings will be retrieved from that file.

There is a companion program called `ShowSTBDatabase` that can be run to display the current users and common settings of the Simulation. This program is executed by entering `tools.simulation.ShowSTBDatabase` on the Simulation command line.

5.3.4 Example Test Applications and Tools

This section shows how to run a simple test application in the simulation. Example test applications can be found in the **nist_ri/devkit/src/testsuite/api** directory. This suite contains simple applications (not Xlets) that exercise specific aspects of the API implementation. These tests can be used to see if you have the simulation and data initialization set up properly. The test suite also provides a way for quickly testing modifications to API implementations. Below is an example of how to run test the SIManager API test.

```
% j gov.nist.hwabstract.JavaSTBMain –r $ROOTDIR/simulation/data/simple.atsc
--simulation output--
> testsuite.api.SIManagerTest
--output from SIManagerTest—
```

The distribution also contains tools that for running Xlets, creating STB state data, and more. The tool RunXlet, for running Xlets is started from the simulation command line as well. See section 6 for details.

6 RUNNING AND CONTROLLING XLETS

6.1 Running Xlets on the Simulation Command Line

Xlets cannot be directly executed from the Simulation command line because they contain no `main()` method, and they have special requirements from the DASE API. Therefore, in order to run Xlets, a native application called `RunXlet` is required. This application is located in package `tools.simulation` and is part of the `devkit.jar` library. In order to execute `RunXlet`, enter `tools.simulation.RunXlet` on the Simulation command line.

`RunXlet` takes the name of the Xlet's entry point class and submits the information on running the Xlet to the Application Registry. From that point on, control of the Xlet can be done via the native Application Selection program. `RunXlet` accepts Xlet names as input until `quit` is entered. All of the Xlet's classes must be located on the `CLASSPATH`.

The `devkit` branch of our tree contains a few Xlets that you can try**. It is advised to feed the Set-Top Box simulation first** (see section 5.3.2) with some data before launching an Xlet by hand. Data files with '.atsc' extension can be found in our tree under '`nist_ri/simulation/data`'. Following is a list of Xlet classnames you can type in the `RunXlet` command line:

- `applications.xlets.simple.ServiceLoopXlet`
- `applications.xlets.weather.WeatherXlet`
- `applications.xlets.stock.StockTicker`
- `applications.xlets.dae.XdmlXlet`

The ServiceLoopXlet is the simplest one and can be used as a starting point to test your setup. The source code for these Xlets can be found in the nist_ri/devkit/src/applications/xlets branch of the distribution tree.

6.2 Running Xlets from a Data Stream

Xlets can be injected into the ATSC data stream along with their data. These Xlets can be auto-started, and are controlled by using the `ApplicationSelection` native application.

In order to build the data stream, use the `DataGen` program located in package `tools.simulation`. This program is intended to be run from the operating system command line, like any other Java program:

```
% java -classpath nist_ri/simulation/runtime/devkit.jar: nist_ri/simulation/runtime/stb.jar DataGen
```

Entering "help" at the `DataGen` command line shows a menu of possible options. In order to create a data stream with an embedded Xlet, you must first create a data service description file. Section 8 describes the `DataGen` program in detail.

6.3 Control of Xlets: Application Selection

The native application `ApplicationSelection` in package `applications.nativeapps.menu` can be used to control Xlets via a graphical interface. This program can be started from the Simulation command line by entering `applications.nativeapps.menu.ApplicationSelection`. Alternatively, `ApplicationSelection` can be launched from the `DtvSimulator` (see section 7) remote by pressing the APPS button.

`ApplicationSelection` allows the user to start, pause, and destroy Xlets. However, if an Xlet that was injected via a data stream is destroyed, it will no longer be available unless it is re-injected with a new Data Service Table update. The simple way to accomplish the update is the inject a different Xlet (and hence a different Data Service Table), and then re-inject the first Xlet.

6.4 Example Xlets

This section gives examples of how to run Xlets from the simulation command line interface. For each example the commands, input, and expected output is given.

6.4.1 ServiceLoopXlet

ServiceLoopXlet is a simple Xlet that you should attempt to run first to verify that the NIST DASE DE is installed correctly and working. ServiceLoopXlet will display the channel name in a HAVi button for each Service in the transport stream data.

```
% j gov.nist.hwabstract.JavaSTBMain –r $HOME/nist_ri/simulation/data/simple.atsc
> tools.simulation.RunXlet
RunXlet  => applications.xlets.simple.ServiceLoopXlet
```

A simulated DTV Display should appear with the name of the Service in a HAVi button in the upper left corner of the screen. The Xlet will loop through the list of Services in the transport stream data displaying each name with a delay between each.

In this example the simulation read data directly from the simple.atsc file given on the command line. The simulation can also be populated with data by using a FIFO. In this manner the simulation can be feed data dynamically. Below is an example of how to feed the simulation data with the FIFO.

In terminal 1:

```
% j gov.nist.hwabstract.JavaSTBMain –f $HOME/inputFIFO
-> tools.simulation.RunXlet
```

In terminal 2:

```
% cat $HOME/nist_ri/simulation/data/simple.atsc > $HOME/inputFIFO
```

In terminal 1:

```
RunXlet => applications.xlets.simple.ServiceLoopXlet
```

6.4.2 Weather X-let

The Weather X-let displays a scrolling weather warning at the bottom of the screen along with a Map and Exit buttons in the upper left corner. Clicking on the Map button will display a weather map. Press Exit to terminate the X-let, including the scrolling warning message. Note that the weather Map and scrolling text are currently static data. See the stock ticker X-let for an example of dynamic data via the Carousel.

```
% j gov.nist.hwabstract.JavaSTBMain
> tools.simulation.RunXlet
RunXlet => applications.xlets.weather.WeatherXlet
```

6.4.3 Stock Ticker X-let

The Stock Ticker Xlet provides scrolling stocks quotes at the bottom of the screen. The Stock Ticker uses the Data Carousel APIs and therefore relies on the JREx functionality described previously in this document.

In terminal 1: (note by default, the fifo is $HOME/inputFIFO)

```
% j gov.nist.hwabstract.JavaSTBMain
-> tools.simulation.RunXlet
```

In terminal 2:

```
% j -classpath $CLASSPATH applications.xlets.stock.StockStreamer -i
$ROOTDIR/simulation/data/quoteList.serial -o $HOME/inputFIFO -pat -t 2000
```

In terminal 1:

```
RunXlet => applications.xlets.stock.StockTicker
```

Note that quoteList.serial contains a list of initial stock quotes that is read by the StockStreamer to create dynamic quote information. The quoteList.serial file is created by the QuoteFileGen utility. If you want to change the initial set of quotes, you'll need to use this utility.

7 THE DTVSIMULATOR APPLICATION

One native application intended for control of the DTV Simulation environment is `DtvSimulator` contained in package `applications.nativeapps.dtvsimulator` of `devkit.jar`. This native application presents a virtual remote control that can be used to launch the native Electronic Program Guide (EPG), the Application Selection program, as well as Simulation setup, such as preferences.

The buttons on the remote control are disabled until the **ON** button is pressed.

The DTV Simulator demonstration consists of a GUI-based DTV Simulation application and GUI-based Stream Feeder and Data Content Display utility programs. The DTV Simulation application has a simulated Remote Control and DTV Display. This section describes how to run the demonstration using the pre-defined transport stream data set.

The table below describes a step-by-step procedure for running the demonstration. The first 3 steps are required and must be performed in the order indicated. Beyond this dynamic control of the demonstration is possible. Terminal refers an X-Term or similar command line window.

Program	**Action**	**Response**
Terminal 1	% j gov.nist.hwabstract.JavaSTBMain –s –t 1	Launch the simulation and automatically start DtvSimulator.
Terminal 2	% j tools.transport.Feeder -t	Launch the Feeder and Data Content GUI utilities and send pre-defined ATSC data into the simulation.
DtvSimulator Remote	Press the ON button	Enable the remote control and turn on the DTV Display. The DTV screen will change from a black background to a picture of a moose. Content Display panel folders will appear with table information.
DtvSimulator Remote	Select actions as described in Table 3 Remote Control Functions. Press the up arrow repeatedly to work through demonstration.	See Table 2 Channel Descriptions and Table 3 Remote Control Functions
Content Display	Click on the folders.	Displays MPEG2 and ATSC table information.

Table 1 Running the DTV Simulator Demonstration

Some channels include Xlets. Table 2 describes what is contained on each channel.

Channel Number	**Scene**	**DASE Application**	**Description**
TNT (2)	Wildlife - Moose	None	None
NEWS (3)	Middle East City	Stock Ticker Xlet	Scrolling stock quotes should appear at the bottom of the screen. **Note that this Xlet requires jreX support.**
TBS (4)	News Room	None	None
NIST (5)	Football Game	Weather Xlet	Scrolling weather warning appears at the bottom of the screen. A Map and Exit button appears in upper left corner. Click on the map button to see weather map. Map should appear below the Map and Exit buttons. Press Exit to terminate the entire Xlet.
CNN (6)	Weather Map	None	None
ABC (7)	Sports Car	HTML Page Presentation	This example demonstrates a DAE application. Click on the “Learn More” button to bring up an HTML browser. The display gives more information about the car. Note that this is not a DASE DAE or DAE application, this

			demonstrates the use of carousel modules for DAE pages and the Application Manager infrastructure necessary for handling DAE applications.

Table 2 Channel Descriptions

NOTE 1: The DTV Simulation application runs very slowly. You may need to wait for repaints of the DTV Screen (or cause repaints to happen manually by hiding and redisplaying windows).

NOTE 2: On the NIST test platforms, running the DTV Simulation application can cause "Out of memory " errors if ran for extended periods. We have not attempted to isolate the memory leak.

Also on the remote control at the bottom are the "Guide", "Apps", "Menu" and "Disp" buttons. These buttons control the Electronic Program Guide, Xlet Monitor Utility Application (Application Selection), User's Preferences Controls, and a Channel recall application. Table 3 describes the functionality of use of these features.

Button	**Description**
On	Turns on the simulation. Loads transport stream data and tunes to channel 2. Activates the other buttons on the remote control.
Off	Disables the all remote control buttons except "On".
Guide	Starts the Electronic Program Guide (EPG) native application. An EPG will transparently overlay the current screen. Click on the "Guide" again to remove the EPG. Double-click on a program listing to get an extended description of the program. Click on the description to return to the EPG.
Apps	Starts the Application Selection native application. See section 6.3 for details.
Menu	Starts the User's Preference native application. Preferences can be used to set favorite channels and more. See the section on User's Preferences for details.
Disp	The "Disp" (Display) button gives detailed program listing information for the channel currently tune to. Click on the "Disp" button again to remove the display.
Up/Down arrows	Used to change channels
Left/Right arrows, Color, PC, M	Not Functional
0-9 Numbers	Can be use to directly tune to a single digit channels (For the demonstration pre-defined data set, channels 2 to 7 are operational).

Table 3 Remote Control Functions

Please note that not all buttons on the remote are functional in the current implementation. The number buttons can be used to directly tune to a channel.

8 THE DATAGEN APPLICATION

`DataGen` is a user interface giving access to a bundle of tools for creating & manipulating bit stream files containing ATSC & MPEG data (.atsc files, see ANNEX B). Practically, it is a simple text interface to manipulate a set of MPEG/ATSC tables & Data Carousel modules. This set sits in an instance `stb.dataypes.ATSCTableSet`. This includes I/O file operations.

Note: for a description of all different files formats manipulated by DataGen, please refer to ANNEX D.

8.1 Syntax

From the shell command line:

```
java tools.simulation.DataGen [-c <command line>]
```

- Without the *-c* option, `DataGen` runs in a loop, waiting for a user command line, executing it and presenting the prompt for another command line. The `HELP` command provides information on running the program and the `QUIT` command exits the program.
- With the -c option, `DataGen` parses & executes `<command line>`. After everything in `<command line>` has been executed, `DataGen` exits. See the file `devkit/Makefile.in` for examples.

8.2 Command line syntax

```
<command line> = <command> [; <command line>]
```

There can be more than one command in the line, separated by semicolons. For example:

```
PRINT ; QUIT
```

prints the memory contents, then quits the program.

Note: commands are **case independent**. We use capitals here for more clarity.

8.3 Commands

8.3.1 Quick overview

Executing the HELP command provides this information:

```
0. { quit | q } --------> Guess...

1. { help | h } --------> Display this help.

2. reset --------------------> Clear memory contents.

3. { exec | e } cmdline ----> Execute a shell command.

4. { readObj | ro } file.obj ---> Fill the memory with the contents of an Object file.

5. { writeObj | wo } file.obj ---> Save the contents of the memory to an Object file.

6. { readATSC | ra } file.atsc ---> Fill the memory by parsing an ATSC bitstream file.

7. { writeATSC | wa } file.atsc ---> Save the contents of the memory to an ATSC
bitstream file

8. { writeMPEG | wm } file.mpeg ---> Save the contents of the memory to a MPEG bitstream
file

9. { importDataService | ids } file.ds ----> Import the data service as described in
file.ds

10. { generateSequence | gs } file.seq outputBaseName -> Generate a test sequence file.

11. { newVirtualChannel | nvc } --------> Add/modify a Virtual Channel.

12. { newEvent | ne } --------> Add/modify an Event in the schedule of a Virtual
Channel.

13. { delVirtualChannel | dvc } --------> Delete a Virtual Channel.

14. { delEvent | de } --------> Delete an Event.
```

15. `{ isolateOneSource | ios } --------> To delete all Virtual Channels but those sharing a specific source_id.`

16. `{ isolateOneTransportStream | iots } ----> To delete all Virtual Channels but those sharing a specific channel_TSID.`

17. `{ rebuildPAT | rpat } --------> Generate a new PAT based on current PMT contents.`

18. `{ rebuildPmtPat | rpp } --------> Rebuild a complete PMT, then generate a new PAT.`

19. `{ newProgram | np } --------> Add/modify a PMT program. Regenerate PAT.`

20. `{ delProgram | dp } --------> Delete a PMT program. Regenerate PAT.`

21. `{ newProgramElement | npe } --------> Add/modify a PMT program element.`

22. `{ delProgramElement | dpe } --------> Delete a PMT program element.`

23. `{ delCarousel | dc } --------> Delete a Data Carousel.`

24. `{ delOrphanCarousels | doc } --------> Delete orphan Data Carousels (i.e. not mentioned in any DST).`

25. `{ delDataService | dds } --------> Delete a Data Service.`

26. `{ changeTSID | tsid } --------> Change TSID value`

27. `print [-o outputFile] -----------> Print memory contents to standard output or to a file.`

8.3.2 RESET

Syntax: `reset`

Action: clear the `ATSCTableSet` currently in memory. All data is lost.

8.3.3 EXEC

Syntax: `{exec | e} cmdline`

Action: pass a command line to the underlying shell and execute it.

8.3.4 READOBJ

Syntax: `{readObj | ro} file.obj`

Action: reset the memory, then fill it with the contents of an Object File (see Annex D.1).

8.3.5 WRITEOBJ

Syntax: `{writeObj | ro} file.obj`

Action: write the memory contents to an Object File (see Annex D.1). Previously existing file is overwritten.

8.3.6 READATSC

Syntax: `{readATSC | ra} file.atsc`

Action: reset the memory, then fill it with the results of parsing the ATSC file.

8.3.7 WRITEATSC

Syntax: `{writeATSC | wa} file.atsc`

Action: write the memory contents to an ATSC File. Previously existing file is overwritten.

8.3.8 WRITEMPEG

Syntax: `{writeMPEG | wm} file.mpeg`

Action: write the memory contents to an MPEG file. Previously existing file is overwritten.

8.3.9 IMPORTDATASERVICE

Syntax: `{importDataService | ids} file.ds`

Action: import a Data Service as described in `file.ds` and incorporate it into existing memory contents by updating and creating MPEG/ATSC tables & Data Carousel modules, as needed.

8.3.10 GENERATESEQUENCE

Syntax: `{generateSequence | gs} file.seq outputBaseName`

Action: this command does not involve nor modify current memory contents. It imports a Sequence as described in `file.seq` and prepares a set of ".atsc" files, one for each step of the sequence where data has to be sent.

Output files: *"`outputBaseName_ch<channel>step<step>.atsc`"* where `<channel>` is a physical channel number and `<step>` is a step number for that channel.

8.3.11 NEWVIRTUALCHANNEL

Syntax: `{newVirtualChannel | nvc}`

Action: add/modify a Virtual Channel in the current Virtual Channel Table. User is prompted for all necessary entries.

8.3.12 NEWEVENT

Syntax: `{newEvent | ne}`

Action: add/modify an event an Event Information in the current Event Information Table. An optional Extended Text description can be added/modified in the current Extended Text Table. User is prompted for all necessary entries.

8.3.13 DELVIRTUALCHANNEL

Syntax: `{delVirtualChannel | dvc}`

Action: remove a Virtual Channel from the current Virtual Channel Table, as well as all corresponding events in the current Event Information Table & Extended Text Table (based on `source_id`). User is prompted for all necessary entries.

8.3.14 DELEVENT

Syntax: `{delEvent | de}`

Action: remove an Event from the current Event Information Table, as well as corresponding description in the Extended Text Table, if any. User is prompted for all necessary entries.

8.3.15 ISOLATEONESOURCE

Syntax: `{isolateOneSource | ios}`

Action: isolate one source by deleting all Virtual Channels in the current Virtual Channel Table but those sharing a specific `source_id` value. All corresponding events are removed from the current Event Information Table and Extended Text Table. User is prompted for `deliverySystemType` and `source_id`.

8.3.16 ISOLATEONETRANSPORTSTREAM

Syntax: `{isolateOneTransportStream | iots}`

Action: isolate one Transport Stream by deleting all Virtual Channels in the current Virtual Channel Table but those sharing a specific `channel_TSID` value. All corresponding events are removed from the current Event Information Table and Extended Text Table. User is prompted for `deliverySystemType` and `channel_TSID`.

8.3.17 REBUILDPAT

Syntax: `{rebuildPAT | rpat}`

Action: reset current Program Association Table and generate a new one based on current Program Map Tables (method `rebuildPMT(...)` of `tools.simulation.ATSCTool`).

8.3.18 REBUILDPMTPAT

Syntax: `{rebuildPmtPat | rpp}`

Action: first update the current Program Map Tables by adding missing Program Elements, based on current Virtual Channel Table contents as well as current Data Service Table & current DSM-CC Data Carousel contents. Second, reset current Program Association Table and generate a new one, exactly as in REBUILDPAT.

Note: all existing Programs and Program Elements are kept.

8.3.19 NEWPROGRAM

Syntax: `{newProgram | np}`

Action: add or modify a program in the current Program Map Tables. Then reset the current Program Association Table and generate a new one, exactly as in REBUILDPAT. User is prompted for all necessary entries.

Note: all existing Programs and Programs Elements are kept.

8.3.20 DELPROGRAM

Syntax: `{delProgram | dp}`

Action: remove an existing program from the current Program Map Tables. Then reset the current Program Association Table and generate a new one, exactly as in REBUILDPAT. User is prompted for a `program_number` value.

8.3.21 NEWPROGRAMELEMENT

Syntax: `{newProgramElement | npe}`

Action: add or modify a Program Element in an existing Program of the current Program Map Tables. The Program Association Table is not modified. User is prompted for all necessary entries.

8.3.22 DELPROGRAMELEMENT

Syntax: `{delProgramElement | dpe}`

Action: remove a Program Element from an existing Program in the current Program Map Tables. Program Association Table is not modified. User is prompted for a `program_number` value.

8.3.23 DELCAROUSEL

Syntax: `{delCarousel | dc}`

Action: remove a Data Carousel from the current repository, that is all modules matching a specific `download_id` value. User is prompted for a `download_id` value.

8.3.24 DELORPHANCAROUSELS

Syntax: `{delOrphanCarousels | doc}`

Action: remove all orphaned Data Carousels from the current repository, that is all Data Carousels not mentioned in any Data Service Table (not indexed by any Tap).

Note: not implemented yet.

8.3.25 DELDATASERVICE

Syntax: `{delDataService | dds}`

Action: remove an existing Data Service Table and all associated resources that are not mentioned in any other Data Service Table (not indexed by any Tap).

8.3.26 CHANGETSID

Syntax: `{changeTSID | tsid}`

Action: change the current Transport Stream ID value (associated with the current data set). User is prompted for a new `tsid` value.

8.3.27 PRINT

Syntax: `print [-o outputFile]`

Action: print all memory contents in a readable manner with hierarchical indentation, either to the standard output or to a file (if `outputFile` specified).

8.4 Assumptions and Simplifications

`These involve both the tools.simulation.DataGen application and the tools.simulation.ATSCByteArrayOutputStream of the devkit.`

- Application resources are conveyed by means of Data Carousel only.
- The resources for a single application are contained in only one data carousel. One data carousel contains resources for only one application: **1 application <=> 1 downloadId**.
- Only one-layer Data Carousel has been implemented so far. Grouping is ignored.
- In a generated Data Service, each Data Service Table Tap points to a single module (not to an entire data carousel).
- One Virtual Channel per `source_id`. This may not always be the case in reality.
- When encoding Data Carousels, the following choices have been made concerning Download Info Indication messages (DIIs):

1. Maximum length of module loop is set to 4046 bytes (case with a `dsmccAdaptationHeader` , 16 bytes for the `dsmccMessageHeader`, no `privateData`, i.e. 4070 bytes for the rest of the `userNetworkMessage()`).

2. The `blockSize` field is set to its maximum value, which is 4066 bytes (`stream_type 0x0B`, see ATSC standard A/90).

3. `transaction_id` is set to the value of the last module processed.

4. All DSM-CC sections are encoded with a CRC32.

5. `dsmccAdaptationHeader` almost always present.

6. 4084 bytes maximum for `userNetworkMessage()`: 14 bytes (12 + 2) for the `dsmccMessageHeader` and 4070 bytes for the rest of the `userNetworkMessage()`.

7. There is no private data: `privateDataLength = 0`.

8. Maximum length of module loop is 4070-24 = 4046 bytes ; `moduleInfoDescription = 0xFF` for "not compressed".

DII: `dsmccAdaptationHeader` almost always present. 4084 bytes max for `userNetworkMessage()`, 14 bytes (12 + 2) for the `dsmccMessageHeader`, 4070 bytes max for the rest of the `userNetworkMessage()`

- No private data: `privateDataLength` must be set to 0
- Max length of module loop is 4070-24 = 4046 bytes
- `blockSize` = max (no real meaning)
- `moduleInfoDescription` must be set to 0xFF to indicate not compressed
- CRC32 is used for the DSM-CC sections
- `transactionId value = value` in the very last module processed

9 THE FEEDER PROGRAM

This is a stand-alone application that simulates low-level native box functions such as tuning & reception. It can be used to use and demonstrate capabilities of both the API implementation and the Set-Top Box simulation. It includes a graphical interface through which the user can choose to send data to the Set-Top Box simulation, as well as browse the content of the data actually sent. This program is located in `tools.transport.Feeder`.

Feeder has a pre-defined set of data files (see the constants in class `ControlFrame`), each one corresponding to a physical channel. The core behavior of Feeder is as follows: Each time the user selects a button, it copies the corresponding data file and sends it to a FIFO file ('output'). Typically the Set-Top Box simulation will read data from 'output'. Feeder also dumps to a FileContentFrame window a human-readable view of the data thus sent.

Optionally, Feeder can receive commands from a 'back-channel' FIFO. Typically, the Set-Top Box simulation can send commands to that FIFO such as "change physical channel to #3" (simulation of a remote control). Detailed description of the 'output' FIFO bit stream syntax can be found in ANNEX B. Detailed description of the 'back-channel' FIFO bit stream syntax can be found in [NEW ANNEX].

Descriptions of the user interface follow:

9.1 Main window (ControlFrame component)

This window includes a menu and a set of buttons. It's always visible. Buttons are used to send data to the 'output'. Most simply send preloaded data, each button corresponding to a different physical channel; whereas the **Stock Feed** button activates/deactivates the generation of random stock quotes on the same physical channel that includes the Stock Ticker data. (Note: Stock Feed is implemented in the `applications.xlets.stock` package).

The menu entry **param** allows the user to set the different file system paths used by the Feeder application to read and write data (i.e. where the FIFO files are). It invokes the Setting component.

The menu entry **mode** permits to switch on and off the back-channel usage. When on, the Feeder application will not only react to user commands (i.e. pressed buttons) but also to send back-channel commands (typically channel switching commands coming from the simulation). When off, back-channel commands are not read (they will accumulate in the back-channel FIFO).

The last menu entry allows exiting the Feeder application.

9.2 Browser window (FileContentFrame & TreeContentPrintStream components)

FileContentFrame is a browsable tree describing all data sent so far to 'output'. It is always visible. Each time a new file is sent, a view of its contents is added at the top level of this tree by the TreeContentPrintStream component. The user can click on any element of the tree to expand or shrink a branch.

9.3 Parameters window (Settings component)

[NOTE: The implementation may not be totally implementing these behaviors]

This window appears only when the user selects the **param** entry in the menu. If allows to change the default values of:

- o **input path**: the path where the set of test data files is. Each time this path is changed, the data files are reloaded.

- **output path**: the path where the 'output' FIFO file should be. Each time this path is changed, the output file is closed then re-opened for the new path.
- **output**: the name of the 'output' FIFO file. Each time this name is changed, the 'output' file is closed then re-opened for the new name.

[back-channel???]

ANNEX A. JAVA RUNTIME ENVIRONMENT EXTENSIONS

The Java Runtime Environment (JRE) extensions implement added functionality to several `java.io` classes. However, because changes were made to some Java source code that is part of the Sun Java Development Kit, these changed files are not delivered as part of the NIST DE. This appendix will describe the changes needed, however.

The following classes have been modified in order to provide the capability to read from Carousel files:

`java.io.FileReader`

`java.io.FileInputStream`

`java.io.RandomAccessFile`

A new class, `CarouselFileConnection` has been added to the `java.io` package. This is a static class that manages a database of `FileDescriptors` for Carousel Files. It really is the Carousel File System: all opened Carousel Files are registered in it. More information about this class can be found in Section A.1.

Finally, a new 'C' library was created in order to still access the native implementation of some `java.io` functions: this library was called `jreX`.

The general idea was to implement in Java a branched treatment of `java.io` functions to replace the previously native implementation. In the case of a regular file, we call the "old" native implementation through the `jreX` library; in the case of a carousel file, the code is present in the function itself. Here is an example for the `read()` method of `java.io.FileInputStream`:

```
public int read() {
     if(fileIsCarouselFile) {
          // FileInputStream instantiated
          // around a carousel file.
          // Java implementation
          // …
     } else  {
          // FileInputStream instantiated
          // around a regular file
          // native implementation
          return readBridge();
     }
}

/* This new native function allows
 * to go around the name conflict
 * and access old native java.io implementation.
 * readBridge is implemented by jreX.c.
 */
```

```
public native int readBridge();
```

With the corresponding 'C' implementation of `readBridge()` in `jreX.c`:

```
/*
 * Class:     java_io_FileInputStream
 * Method:    readBridge
 * Signature: ()I
 */

JNIEXPORT jint JNICALL Java_java_io_FileInputStream_readBridge
(JNIEnv * env, jobject thisObj) {
     return Java_java_io_FileInputStream_read(env, thisObj);
}
```

The same scheme was applied to all native methods. A detailed description of all necessary changes can be found in the source code tree under `ext/jreX/javalib/changes/`. The changed versions of the `java.io` Java source files should be placed in the ext/jreX/javalib/src/java/io directory in order for the build system to find them. The build system will attempt to incorporate the changed files into the NIST DASE environment after running the command '`configure –with-jrex`' from the top-level directory. Consult the README file in the top-level directory for instructions in performing the build.

A.1 Details of the CarouselFileConnection Class

The role of `java.io.CarouselFileConnection` is to implement a carousel file system in addition to the existing regular file system. This means managing a database linking each opened `CarouselFile` to its `FileDescriptor`. This `FileDescriptor` is from the `java.io` package, following the DASE API design that implies uniform access to both disk files and carousel files. `CarouselFileConnection` has package visibility only and was designed in two parts:

- A static part with package visibility that manages a list of open connections: A hash table of {`FileDescriptor`, `CarouselFileConnection`} entries.
- A non-static part with private visibility that manages 1 or more opened connections to a specific `CarouselFile`.

The static part offers an interface to other members of the `java.io` package, namely `FileInputStream`, `FileReader` and `RandomAccessFile`. This interface will typically be called when the regular `java.io` classes need to deal with `CarouselFile` instances. Functions include creating a `CarouselFileConnection`, adding/removing users to a `CarouselFileConnection` and `getXXX()` functions to access the file. Creating a `CarouselFileConnection` simply means creating a new `FileDescriptor` and associating it to the `CarouselFile` through a new `CarouselFileConnection` instance.

The non-static part is private and represents one opened file with one or more users. A `CarouselFileConnection` instance whose number of users falls back to zero is automatically removed from the static table.

ANNEX B. BITSTREAM SYNTAX

In the early steps of the DASE API implementation, ATSC PSIP tables were needed for testing. Since no data was available on the air, we had to build our own test data. We needed ATSC PSIP tables only and were not dealing with any real-time issue, hence decided to avoid most of the MPEG encoding/decoding by encoding the MPEG payload only, that is the ATSC PSIP tables.

A simplified bit stream format called '.atsc' format was therefore created. Its basic component is one packet, with a short header (a dozen of bytes) and a payload carrying one entire PSIP table. Later, as needed by the tests, other type of data were added: Data Carousel modules, acknowledgement ('ACK') messages for box-native actions such as shifting channels from the remote control, and PCR messages.

PCR messages were added when we started using an underlying parser (C program) that would parse actual MPEG files and transmit data to different modules including the simulation.

ACK messages were added when we started simulating channel shifting through multiple physical channels with a Feeder program. This Feeder program uses a pre-defined set of '.atsc' files, one for each channel (in the tree: `devkit/src/tool/transport/GFeederApplication`).

Here is the pseudo-code bit stream syntax of one '.atsc' packet (header + payload):

```
16/24 bytes header
-----------------------

fifoHeader {
  headerType               8 bits         // data or table

  if (headerType == TABLE) {

    table_id             8 bits
    reserved             3 bits = 0
    pid                 13 bits
    table_id_extension  16 bits
    table_type          16 bits
    reserved            12 bits = 0
    tableLength         20 bits
    reserved            32 bits = 0

  } else if (headerType == DATA) {

    dataId           8 bits         // DSM-CC Carousel only for now
    reserved             3 bits = 0
    pid                 13 bits
    downloadId          32 bits
    reserved             8 bits = 0
    moduleVersion        8 bits
    moduleId            16 bits
    moduleSize          32 bits

  } else if (headerType == ACK || headerType == NACK) {

    command              8 bits
    if (command == CHANGE_CHANNEL) {
      newChannel        16 bits
    } else {
      reserved           3 bits = 0
      targetPid         13 bits
    }
    reserved            96 bits = 0
  } else if (headerType == PCR) {

    adaptationFlags      8 bits
    reserved             3 bits = 0
    pid                 13 bits
    reserved            22 bits = 0
    if (adaptationFlags & 0x10 == '1') {
      pcr               42 bits
    } else {
      reserved          42 bits = 0
    }
    reserved            32 bits = 0

  }
}

if (fifoHeader.headerType = DATA && fifoHeader.dataId = DATA_CAROUSEL) {
  reserved              23 bits = 0
  pts_is_valid           1 bit
  reserved               7 bits = 0
  pts                   33 bits
}

fifoData {
  // tableLength or moduleSize bytes of data
}
```

CONSTANTS:

----

headerType: 0x01 -> TABLE

0x02 -> DATA

0x21 -> ACK

0x22 -> NACK

0x23 -> PCR

other values are reserved for future use

dataId: 1 -> DSM-CC Data Carousel

other values are reserved for future use

command: 0x31 ("1") -> CHANGE_CHANNEL

0x32 ("2") -> OPEN

0x33 ("3") -> START

0x34 ("4") -> STOP

0x35 ("5") -> CLOSE

ANNEX C. PLATFORM INSTALLATION NOTES

C.1 Summary of supported Platforms

In general, only java version 1.2 and higher can be used since the Set-Top Box simulation is using features present in 1.2 but not in the pJava defined in DASE-1. Also, native threads are preferred to green threads since the Data Carousel API is implemented with JNI.

The table below summarizes our experience of running JavaSTBMain and the Feeder on various Linux, Solaris, and Windows NT platforms.

Platform	JDK	Outcome
Linux	jdk 1.2.2rc4 native threads (Blackdown implementation)	Successful
Linux	jdk 1.2.2_008 green threads	JavaSTBMain hangs during initialization process; Feeder works
Linux	jdk 1.3.0	Successful
Red Hat Linux 7.1/Intel P4	jdk 1.3.1	Successful
Red Hat Linux 7.2/Intel P3	jdk 1.3.1	Successful
Solaris 2.7/Intel	jdk 1.2.2_05	Successful
Windows NT 4.0	jdk 1.3.1	Successful, except for jreX extensions.

C.2 Linux Notes

The Simulation has been compiled and run successfully on a Red Hat 7.1 system with JDK version 1.3.1 available from Sun. Also, in order to compile the Simulation, you'll need a JMF library. Sun hasn't released JMF for Linux, but you can copy the `jmf.jar` file from the Solaris JMF distribution to the `/usr/jmf/lib` directory on your Linux workstation. (At this time, the Simulation doesn't invoke the JMF calls, so the remaining JMF libraries are not needed; only the class files).

The Simulation has been compiled and run on a Red Hat 6.0 system with the JDK version 1.2.2 available from Sun. In order to run the Simulation on Red Hat 6.0, our experience shows that the native threads library works best. Therefore, you will need to have the path to this library in your `LD_LIBRARY_PATH` environment variable. The following directories need to be added to `LD_LIBRARY_PATH`:

```
/usr/java/jre/lib/i386
/usr/java/jre/lib/i386/classic
/usr/java/jre/lib/i386/native_threads
```

C.3 Windows NT Notes

The Windows NT instructions assume you have the Cygwin Tools 1.1.x or higher installed. These tools provide Unix utilities that ease the porting. The Unix tools include gcc, Makefile, and the bash shell.

As with Solaris and Linux, certain environment variables need to be set before running configure. (The following assumes the use of a bash compatible shell for environment variable syntax)

`Note:` `the use of the forward “/” slashes for setting these variables.`

```
JAVA_HOME=//C/jdk1.3
JMFHOME=//C/jmf1.1
export JAVA_HOME
export JMFHOME
```

The CLASSPATH can be set as follows and needs to be set properly before compiling or running. The CLASSPATH is needed for running any component of the simulation, for example JavaSTBMain and Feeder. Thus, since these components are often started from separate windows, it is advisable that the CLASSPATH be set either in each window, or in the global Windows environment variable settings dialog box)

Important Note: the use of the backward "\" slashes for settings these variables.

```
ROOTDIR="C:\nist_ri"
export ROOTDIR
CLASSPATH="${ROOTDIR}\simulation\runtime\stb.jar;${ROOTDIR}\simulation\runtime\d
ase.jar;${ROOTDIR}\simulation\runtime\devkit.jar"
export CLASSPATH
```

ANNEX D. FILE FORMATS USED BY DATAGEN

Filenames have a preferred extension for each format. Theses extensions are not mandatory at all, but they are the ones used for the data files delivered with the NIST RI.

For each type, associated commands in tools.simulation.DataGen are given (see section 0).

D.1 OBJ File

Preferred extension: .obj

Associated commands: READOBJ, WRITEOBJ

Description: this file format defines a complete representation of **one** *stb.datatypes.ATSCTableSet* instance, written through a *java.io.ObjectOutput* (see *tools.simulation.ATSCTool.writeToObjectOutput()*).

D.2 ".atsc" File

Preferred extension: .atsc

Associated commands: READATSC, WRITEATSC

Description: this file format defines a bit stream that carries a subset of the information contained in a MPEG bit stream (see ANNEX B). It was defined & used to feed the Set-Top Box Simulation with test data. It contains:

- MPEG and ATSC tables.

- Data Carousel modules.
- ``hardware" notifications such as ``physical channel was successfully changed to number 57". Practically these messages come from an application that simulates the hardware (*tools.transport.Feeder*).

Associated classes: such a bit stream can be encoded using methods from *tools.simulation.ATSCByteArrayOutputStream* and parsed using a *stb.managers.ATSCByteStreamParser* object.

Associated user applications: a user can create an “.atsc" bit stream file in two ways:

- From scratch, editing tables & data carousel using the *DataGen* application.
- From an MPEG bit stream, using a C parser.

Detailed syntax: see ANNEX B.

D.3 MPEG File

Preferred extension: .mpeg

Associated command: WRITEMPEG

Description: this file format defines a standard MPEG bit stream.

Associated class: *tools.simulation.MPEGWrapper* is a filter that encapsulates a “.atsc" bit stream (see ANNEX B) into a MPEG bit stream.

D.4 DS File

Preferred extension: .ds

Associated command: IMPORTDATASERVICE

Description: this file format defines in a human-readable way **one** Data Service, as defined by ATSC standard A/90.

Syntax: ***the syntax is case-sensitive!*** The unit is a line. The lines can be written in any order. The global syntax of a line is defined by the first word on the line:

- if the first word of a line starts with '#', it is a comment line and will be ignored by IMPORTDATASERVICE.
- if the first word of a line is 'dataService', the line will define the attributes of the Data Service.
- in all other cases, lines are grouped that have the exact same first word. Each group describes one application of the Data Service.

Example:

```
# This line starts with a '#' and is therefore a comment
# Data Service attributes: lines start with the "dataService" word
# Data Service name (goes in DST ServiceInfo descriptor loop).
dataService name theDataServiceName
# Def. of an app: lines are linked together by the "app1" word
# Application AppId:
#
# <appWord> appId { DASE <providerAuthority> <providerIndex> <appName> <appVersion> | UN
KNOWN <appIdBytes> }
#
# Where:
# <appWord> identifies the app., within the context of this .ds file
# <providerAuthority> is an hex value, e.g. d4e76590
# <providerIndex> . . . . . .
# <appName> is a string
# <appVersion> is an array of 8-bit hex values, e.g. 23 bf 4c
#
# Example:
app1 appId DASE
# Application downloadId (hex form).
# NOTE: if you plan on having multiple Data Services coming sequentially
# on a program, you should make sure downloadIds are different.
app1 downloadId 12345678
# Application profile & level (1 byte for each, hex form).
app1 profile FE 7A
# Application title
app1 title theApplicationTitle
# Application taskScope & taskPriority (1 byte for each, hex form).
app1 task 01 11
# OPTIONAL: indicates that the Application should be automatically
# launched when received.
# -> when autoLaunch, all Taps get their systemStateFlag set to "Bootstrap"
# (0x01) instead of "Runtime" (0x00)
app1 autoLaunch
# AT LEAST ONE: Application entryPoint class name (Xlet).
# As defined by S13, an Application can contain multiple entryPoints ;
# not to be confused with a DASE Application = 1 Xlet.
# -> one S13 App = many DASE Apps
#
# Syntax: <appWord> entryPoint <className>
app1 entryPoint mypackage.MyXlet
```

```
# Application list of module files, including the entryPoints ones.
# Syntax for one module:
#
# <appWord> module <fileName> [c | classPath] [e | entryPoint] [n | name <name>]
# [contentType <contentType>] [version <vNumber>]
# [ {be | bigEndian} | {le | # littleEndian} | {ue | unspecifiedEndian} ]
#
# Where:
# <appWord> identifies the app, within the context of this .ds file
# <fileName> (string) the path of the file containing the data
# <name> (string) the full classname (including packages)
# <contentType> (string) the MIME type of the data carried by the module
# <vNumber> (hex, 1 byte) the version number of the module
#
#
# bigEndian, littleEndian & unspecifiedEndian define the value of the
# moduleEndian field (see DII in S13). unspecifiedEndian is the default
# value.
#
app1 module theFileName c e n thepackagename.TheClassName contentType application/java v
ersion 1
app1 module anotherFileName c e name thepackagename.AnotherClassName contentType applica
tion/java version 1
```

D.5 SEQ File

Preferred extension: .seq

Associated command: GENERATESEQUENCE

Description: this file format defines in a human-readable way a test sequence, generally used to produce a bitstream. This includes timing and different file formats such as ``.atsc", OBJ & DS. It can be used by *tools.transport.Feeder*.

Example:

```
including all possibilities:
# Sequence file to test the simulation
1 atsc nist20_with_events.atsc
wait 5
2 ds BoxScore.ds
wait 10
3 obj 05mo.obj
wait
```

```
2 ds BoxScoreAndTest.ds
wait 5
2 ds Test.ds
wait 8
1
```

Below, each line is explained, as far as what a program executing such a sequence should do.

```
# Sequence file to test the simulation
```

A line is ignored if empty or the first character is '#'.

```
1 atsc nist20_with_events.atsc
```

Change to physical channel 1 (if necessary). Send an atsc file named ``nist20_with_events.atsc''.

```
wait 5
```

Wait 5 seconds before executing next line.

```
2 ds BoxScore.ds
```

Change to physical channel 2. Read a data source file named ``BoxScore.ds'', encode it in ``.atsc'' format and send it.

```
wait 10
```

Wait 10 seconds before executing next line.

```
3 obj 05mo.obj
```

Change to physical channel 3. Read an object file named "05mo.obj", encode it in atsc format and send it.

```
wait
```

Wait for a user input.

```
1
```

Switch to physical channel 1.

ANNEX E. EXAMPLE SETUP ENVIRONMENTS

Following is an example of environment setup (for example, to put in a `.bashrc` file). In this example, we're using JMF 1.1 installed as `/usr/local/jmf/lib/jmf.jar`. Any later version of JMF also works, but it is advised to compile against JMF 1.1 since it is the version specified by the DASE-1 standard.

Notes:

- The 'j' alias (see examples below) for the 'java' command: it is necessary for using jreX extensions, which implement the Data Carousel API (see section 4.3). It uses a non-standard JVM option: `'-Xbootclasspath'`. Indeed, trying to set directly the `sun.boot.class.path` property with the `'-D'` option didn't prove to be effective before the end of the JVM boot process.
- The `'JMFHOME'` variable indicates where JMF is installed. The minimum install for JMF is to put the `'jmf.jar'` file, version 1.1, under `$JMFHOME/lib`/.

E.1 CSH Shell Example

The following is an example of additional environment variables and aliases that should be included in the .cshrc file.

```
####################################
# Sample Environment for NIST DASE DE
####################################

setenv JAVA_HOME /usr/java

setenv JMFHOME /usr/jmf

setenv HOME /export/home/test

setenv ROOTDIR ${HOME}/nist_ri

setenv RUNTIME ${ROOTDIR}/simulation/runtime

setenv JAVA_ARCH i386

setenv  CLASSPATH "${RUNTIME}/stb.jar:${RUNTIME}/devkit.jar:${RUNTIME}/dase.jar:${JMFHOME}/lib/jmf.jar"

alias j 'java -Xbootclasspath:${RUNTIME}/jreX.jar:${JAVA_HOME}/jre/lib/rt.jar:${CLASSPATH}  -Dsun.boot.library.path=
${JAVA_HOME}/jre/lib/i386:${RUNTIME}'
```

E.2 BASH Shell Example

```
####################################
# Sample Environment for NIST DASE DE
####################################

# For JMF, the minimum install is to put the jmf.jar file, version 1.1,
# under $JMFHOME/lib

JMFHOME=/usr/local/jmf
JAVA_HOME=/usr/java
JAVA_ARCH=i386

ROOTDIR=${HOME}/nist_ri
RUNTIME=${ROOTDIR}/simulation/runtime

export JMFHOME JAVA_HOME JAVA_ARCH ROOTDIR RUNTIME

# settings specific to the use of jreX extensions
# set java.class.path

CLASSPATH=${RUNTIME}/stb.jar:${RUNTIME}/devkit.jar:${RUNTIME}/dase.jar:${JMFHOME}/lib/jmf.jar
export CLASSPATH

alias j ='java -Xbootclasspath:${RUNTIME}/jreX.jar:${JAVA_HOME}/jre/lib/rt.jar:${CLASSPATH} -Dsun.boot.library.path=
${JAVA_HOME}/jre/lib/i386:${RUNTIME}'
```